# Agent surgery: The case for mutable agents

Ladislau Bölöni and Dan C. Marinescu
Computer Sciences Department
Purdue University
West Lafayette, IN 47907

August 30, 2005

**Abstract**

We argue that mutable programs are an important class of future applications. The field of software agents is an important beneficiary of mutability. We evaluate existing mutable applications and discuss basic requirements to legitimize mutability techniques. Agent surgery supports runtime mutability in the agent framework of the Bond distributed object system.

## 1 Introduction

Biological metaphors from structural biology, immunology, and genetics provide useful insights for the design of complex systems. For example a component-based design inspired by protein structure has distinct advantages over more traditional approaches [5], [6]. In this paper we continue this line of thought and discuss another metaphor, program mutability.

Introductory computer architecture courses often present an example of self-modifying assembly code to drive home the message that executable code and data are undistinguishable from one another once loaded in the internal memory of a computer. Later on, computer science students learn that writing self-modifying programs or programs which modify other programs is not an acceptable software engineering practice but a rebellious approach with unpredictable and undesirable results. A typical recommendation is *"Although occasionally employed to overcome the restrictions of first-generation microprocessors, self-modifying code should never ever be used today - it is difficult to read and nearly impossible to debug."* [1]

While giving the proper consideration to this point of view, in this paper we argue that self-modifying or mutable programs are already around, and they will be an important component of the computing culture in coming years. In particular, this approach can open new possibilities for the field of autonomous agents.

Software agents play an increasingly important role in today's computing landscape due to their versatility, autonomy, and mobility. For example the Bond agent framework discussed briefly in Section 3 is currently used for a variety of applications including an workflow enactment engine for commercial and business applications [9], resource discovery and management in a wide-area distributed object-system, [11], adaptive video services [10], parallel and distributed computing [7]. Some of these applications require that the functionality be changed while the agent is running. But to change the functionality of an agent we have to modify its structure. This motivates our interest for agent surgery and by extension for program mutability.

Program mutability is often motivated by the need to extend the functionality of a running application. Probably the best known examples of this case are the plugins allowing a Web browser to accept new data types. Another reason to modify a running application is to minimize the level of resource consumption. For example, to reduce the latency and bandwidth requirements for migrating an agent from one location to another we may only want to transport the new site only those components of the agent that will be activated in the future rather than all the components.

This paper is structured as follows. In Section 2 we identify the features of various existing or envisioned mutable programs and propose a distinction between weak and strong mutability. In Section 3 we present the self-modifying architecture of the agent framework of the Bond distributed object system, while in Section 4 several common surgery techniques are discussed. Section 5 presents a real-world application of agent surgery. Conclusions are drawn in section 6.

## 2 Mutable programs

We propose the term *mutable programs* for cases when the program executable is modified by the program itself or by an external entity. We use the term "mutable" instead of "mutating" to indicate that changes in the program structure and functionality are directed, well specified, non-random. The word mutation is used in genetic algorithms and evolutionary computing to indicate random changes in a population when the "best" mutant in some sense is selected, using a survival of the fittest algorithm. In these cases the source of the program is modified.

The case we consider is when a program is modified at run time to change its functionality. The user of the program is usually (but not always) asked for permission or at least notified about the modification. The modification is initiated by the user, the program itself, or by an external program. Self-modifying programs are a subset of mutable programs. We distinguish *weak* and *strong* mutability.

**Weak mutability** is the technique to *extend* the functionality of the application using external components.The application still keeps its essential characteristics and functionality. The trademark of weak mutability is a well defined API or interface between the program to be extended and the entity providing the additional functionality. The mechanism of extension is either the explicit call of an external application or loading of a library, either in the classical dynamic library sense or in the Java applet / ActiveX control sense.

This is the currently most accepted form of mutability in applications. Examples include:

-**data format plugins**, to extend the functionality of an application, enabling it to handle new data formats e.g. plugins of web browsers like Netscape.

-**active plugins**, to provide new processing methods based upon a standard API, e.g. plugins of image processing applications like Adobe Photoshop, or The Gimp.

-**skins and themes**, to change the look and feel of applications (e.g. WinAmp) or entire desktops or operating systems. While these are sometimes just changes in pixmaps like in the WinAmp or Windows themes case, in other cases they require the change of the drawing and manipulation code of the entire system e.g. the KDE styles.

-**applets and embedded applications**, to provide new functionality of an application by embedding new active components which respect a well defined interface. These applications are usually isolated from the original application. For example, the interaction between a Java applet and the rest of the browser, or indeed the operating system is practically non existant.

-**automatic upgrades** where parts of application are replaced with a never or different version.

The frontiers of weak mutability are blurred. Most programs are assembled from a number of dynamic libraries, and upgrading those libraries is a common practice. Various criteria can be proposed to differentiate between a static and a mutable program.

In case of **strong mutability**, applications change their behavior in a radical manner. The APIs for the modification is loosely defined, or implicit.

Strong mutability can be implemented at any level of granularity. At the machine-code level, typical example are viruses and anti-virus programs. Classical executable viruses are attaching themselves to programs and modify the loader code to add their additional, usually malign, functionality. Certain viruses perform more complex procedures on the executable code, e.g. compression, redirection of function calls etc. Anti-virus programs modify the executable program, by removing the virus and restoring the loader. A special kind of anti-virus programs, called vaccines modify an executable to provide a self-checksum facility to prevent further modifications.

Another example of strong mutability is *code mangling* [2], performed to prevent the reverse engineering of the code. A somewhat peculiar example is the case of self-building neural networks [3].

Component-based systems are most promising for strong mutability because the larger granularity of the components makes the problem more tractable. At the same time the cleaner interface of components and the fact that they are usually well specified entities allow for easier modification. The most promising directions are the custom assembly of agents based on specifications, and runtime modification of component-based agents, called in this paper agents surgery. An example of the latter is the Bond agent system discussed in section 3.

An example of application-level strong mutability is the case of dynamically assembled and modified workflows [4].

## 2.1 Requirements and mechanisms for mutability

Several requirements must be met before mutabilility could become an accepted programming technique, they are: well-defined scope, self-sufficiency, seamless transformation, persistency, and reversibility.

First, the modification should have a **well-defined scope** and lead to a predictable change of program behavior. For example a user downloading a new plugin expects that the only modification of the program behavior is to accept a new data type. If a Bond agent adds a new plane for getting online news or for watching the stock market other aspects of agent's behavior should not change.

**Self-sufficiency** requires that the change described by the process should be defined without knowing the internal structure of the application. Thus multiple consecutive changes can applied to an application. It also allows one change to be applied to multiple applications. For example, we can design a surgical blueprint to replace the 64 bit RSA encryption with a 128 bit IDEA encryption in a Bond agent. This blueprint can be applied to all agents in an organization regardless of their internal structure or the fact that they were surgically altered before.

**Seamless** transformation of program behavior, the change should be performed on a running program without the need to stop its execution. The alternative to modify the source code, compile it, and restart the program is not acceptable in many cases. As our dependency upon computers becomes more and more pronounced, it will be increasingly difficult to shut down a complex system e.g. the air-traffic control system, a banking system, or even a site for electronic commerce to add, delete, or modify some of its components. Even today it is not acceptable to restart the operating system of a computer or recompile the browser to add a new plugin necessary to view a webpage. A similar hardware requirement is called hot-plug compatibility, hardware components can be added and sometimes removed from a computer system without the need to shut down the system.

Another requirement is to make the change **persistent**. This requirement is automatically satisfied when the file containing the executable program is modified. If the image of a running program is modified then a mechanism to propagate this change to the file containing the executable must be provided.

The reverse side of the coin is the ability to make the change **reversible**. In a stronger version of this requirement we expect to revert a change while the program is running. A weaker requirement is to restart the agent or the application in order to revert the change. This requirement can be easily satisfied for individual changes, by keeping backup copies of the original program. If we allow multiple changes and then revert only some of them the problem becomes very complex as the experience with **install programs/scripts** shows.

We emphasize that not all these requirements should be satisfied simultaneously. Actually only the predictability is a critical property. Satisfying the additional requirements however, broadens the range of applicability of the technique. Thus, when designing the **mechanisms of mutability** we should attempt to satisfy as many of them as possible.

The common feature of every mechanism of mutability is that we treat code as a data structure. This is an immediate consequence of the von Neumann's concept of stored-program computers, and thus applicable to virtually any program. Yet, in practice, modifying running programs is very difficult, unless they are described by a simple and well-defined defined data structure accesible at run-time. For example, a program written in C++ or C has a structure given by the flow of function calls in the original source code. After the compilation and optimization process however, this structure is very difficult to reconstruct.

The object code of compiled languages or the code generated from the assembly language, while it can be viewed as a data structure, it does not allow us to easily discover its properties. This complexity justifies the point made at the beginning of this paper, that self-modifying machine code should not be used as a programming technique.

In conclusion, programs can be successfully modified if there is a high level, well documented data structure [1] that in some sense is analogous with the genetic information of biological structures.

[1] Of course, strong mutability is possible even without this high level data structure, if the external entity is able to figure out the low level data structure manifests itself at the object code level - viruses are doing exactly this. However, this cannot form the basis of a generally viable technique.

If the program designer chooses to have only part of the code described by this structure (like in the case of plugin API-s) weak mutability is possible. If the entire program is described by the data structure then strong mutability is possible. This is the case of Bond agents whose behavior is based on the multiplan state machine and the agent surgery enables strong mutability.

# 3 Agent surgery - a mechanism for mutability

## 3.1 The Bond distributed object system

Bond is a Java based distributed object system and agent framework, with an emphasis on flexibility and performance. It is composed of a core level, containing the object model and message oriented middleware, a service layer containing distributed services like directory and persistent storage services and the agent framework, providing the basic tools for creating autonomous network agents together with a database of commonly used strategies which allow developers to assemble agents with no or minimal amount of programming.

**Bond Core.** At the heart of the Bond system there is a Java Bean-compatible component architecture. Bond objects extend Java Beans by allowing users to attach new properties to the object during runtime, and offer a uniform API for accessing regular fields, dynamic properties and JavaBeans style `setField/getField`-defined virtual fields. This allows programmers the same flexibility like languages like Lisp or Scheme, while maintaining the familiar Java programming syntax.

Bond objects are network objects by default: they can be both senders and receivers of messages. No post-processing of the object code as in RMI or CORBA-like stub generation, is needed. Bond uses *message passing* while RMI or CORBA-based component architectures use *remote method invocation*.

The system is largely independent from the message transport mechanism and several communication engines can be used interchangeably. We currently provide TCP-based, UDP-based, Infospheres-based, and, separately, a multicast engine. Other communication engines will be implemented as needed. The API of the communication engine allows Bond objects to use any communication engines without the need to change or recompile the code. On the other hand, the properties of the communication engine are reflected in the properties of the implemented application as a whole. For example the UDP based engine offers higher performance but does not guarantee reliable delivery.

All Bond objects communicate using an agent communication language, KQML [15]. Bond defines the concept of *subprotocols*, highly specialized, closed set of commands. Subprotocols generally contain the messages needed to perform a specific task. Examples of generic Bond subprotocols are *property access* subprotocol, *agent control* subprotocol or *security* subprotocol. An alternative formulation would be that subprotocols introduce a *structure in the semantic space of the messages*.

Subprotocols group the same functionality of messages which in a remote method invocation system would be grouped in an *interface*. But the larger flexibility of the messaging system allows for several new techniques which are difficult to implement in the remote method call case:

- The subprotocols implemented by objects are properties of the object, so two objects can use the property access subprotocol implemented by every Bond object, to find the common set of subprotocols they can use to communicate.
- An object is able to control the path of a message and to delegate the processing of the message to subcomponents called *regular probes*. Regular probes can be attached dynamically to an object as needed.
- Messages can be intercepted before they are delivered to the object, thus providing a convenient way to implement security by means of a fire wall, accounting, logging, monitoring, filtering or preprocessing messages. These operations are performed by subcomponents called *preemptive probes* which are activated before the object in the message delivery chain.
- Subprotocols, like interfaces, are grouping some functionality of the object, which may or may not be used during its lifetime. A subcomponent called *autoprobe* allows the object to instantiate a new probe, to handle an incoming message which can not be understood by the existing subcomponents attached to an object.

- Objects can be addressed by their unique identifier, or by their *alias*. Aliases specify the services provided by the object or its probes. An object can have multiple aliases and multiple objects can be registered under the same alias. The latter enables the architecture to support *load balancing* services.

These techniques can be implemented through different means in languages which treat methods as messages, e.g. Smalltalk. In Java and C++ they can be implemented at compile time, not at runtime, e.g. using the delegation design pattern. Techniques from the recent CORBA specifications e.g. the simultaneous use of DII, POA, trading service and others, also allow to implement a similar functionality, but with a larger overhead, and significantly more complex code.

**Bond Services.** Bond provides a number of services commonly used found in distributed object systems, like directory, persistent storage, monitoring and security. Event, notification, and messaging services, which provide message passing services in remote method invocation based systems are not needed in Bond, due to the message-oriented architecture of the system.

Some of Bond services perform differently than their counterparts in other middleware systems, like CORBA. For example, Bond never requires explicit registration of a new object with a service. Finding out the properties of a remote object, i.e. the set of subprotocols implemented by the object, is done by direct negotiation amongst the objects. The directory service in Bond combines the functionality of the naming and trading services of other systems and it is implemented in a distributed fashion. Objects are located by a search process which propagates from local directory to local directory. The directories are linked into a virtual network by a transparent *distributed awareness* mechanism, which transfers directory information by piggybacking on existing messages as discussed in the previous Section.

Compared with the naming service implementations in systems like CORBA or RMI, which are based on the existence of a name server, this approach has the advantage that there is no single point of failure, and the distributed awareness mechanism reconstitutes the network of directories even after catastrophic failures. However, a distributed search can be slower than lookup on a server, especially for large networks of Bond programs. For these cases, Bond objects can be registered to external directories, either to a CORBA naming service through a gateway object, or to external directory services using LDAP access.

## 3.2 Bond Agents

The *Bond agent framework* is an application of the facilities provided by the Bond core layer to implement collaborative network agents. Agents are assembled dynamically from components in a structure described by a multi-plane state machine, [12]. This structure is described by a specialized language called `blueprint`. The active components (*strategies*) are loaded locally or remotely, or can be specified in interpretive programming languages embedded in the blueprint script. The state information and knowledge base of the agents are collected in a single object called *model of the world* which allows for easy checkpointing and migration of agents. The multiplane state machine describing the behavior of agents can be modified dynamically, thus allowing for *agent surgery*.

The *behavior* of the agent is described by the *actions* the agent is performing. The actions are performed by the strategies either as reactions to external events, or autonomously in order to pursue the *agenda* of the agent. The current state of the multiplane state machine (described by a *state vector*) is specifying the strategies active at a certain moment. The multiple planes are a way of expressing parallelism in Bond agents. A good technique is to use them to express the various facets of the agents behavior: sensing, reasoning, communication/negotiation, acting upon the environment and so on. The *transitions* in the agent are modifying the behavior of the agent by changing the current set of active strategies. The transitions can be triggered by internal events or from external messages - these external messages form the *control subprotocol* of the agent.

Strategies, having limited interface requirements are a good way to provide code reuse. The Bond agent framework provides a strategy database, for the most commonly used tasks, like starting and controlling external agents or legacy applications. A number of base strategies for common tasks like dialog boxes or message handlers are also provided, which can be sub-classed by developers to implement specific functionality. External algorithms, especially if written in Java are usually easily portable to the strategy interface.

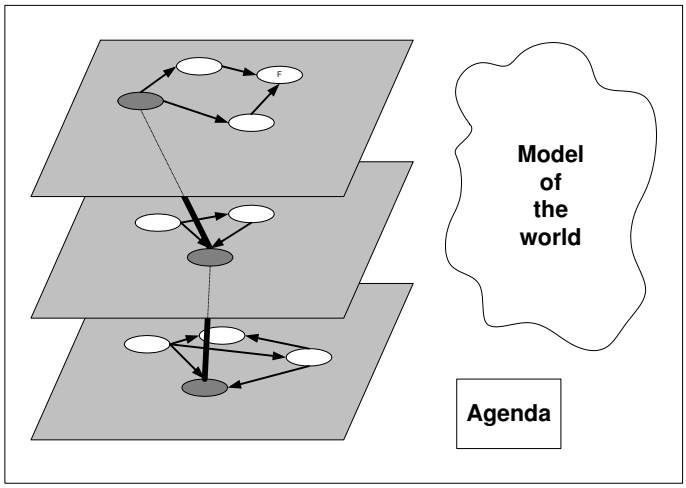


Figure 1: The multiplan structure of a Bond agent

### 3.3 Mutability in Bond

Agent surgery is a modification technique employed in the Bond system, through which the behavior of the *running* agent is changed by modifying the multiplane state machine which describes its behavior. In section 2.1 we have identified that at the basis of any mutable program stays a certain data structure describing its code. For agent surgery, this data structure is the multiplane state machine.

There are several advantages the multiplane state machines offer over other data structures:

- The behavior of the agent in any moment is determined by a well defined subset of the multiplan state machine (the current state vector). Most states are inactive which allows for any surgical operation which does not affect the current state to be performed during run-time.
- The multiplane state machine exhibits *enforcable locality*. While most programs exhibit the statistical locality property (which is the basis of all cache-like hardware and software solutions), this can not be used in mutability procedures, because we don't have a guarantee that the program will not perform suddently a long jump, which can be intercepted only at a very low level - by the operating system or the hardware. The semantic equivalent of a long jump, the transitions, however, are executed through the messaging function of the agent, thus they can be captured and queued for the duration of the agent surgery.
- The multiplane state machine is *self-describing*. It's structure can be iterated on by an external component. This allows to make the changes persistent by allowing to write out the runtime modifications to a new blueprint script. On the other hand allows for specifying operations independently from the structure of the agent.

Bond agents can be modified using "surgical" `blueprint` scripts. In contrast with the scripts used to create agents which define the structure of a multiplan finite state machine, surgical blueprints are adding, deleting or changing nodes, transitions and/or planes in an existing multiplan finite state machine.

The sequence of actions in this process is:

(1)A *transition freeze* is installed on the agent. The agent continues to execute normally, but if a transition occurs the corresponding plane is frozen. The transition will be enqueued.

(2) The agent factory interprets the blueprint script and modifies the multi-plane state machine accordingly. There are some special cases to be considered: (a) If a whole plane is deleted, the plane is brought first to a soft stop - i.e. the last action completes. (b) If the current node in a plane is deleted, a *failure* message is sent to the current plane. If there is no failure transition from the current state, the new state will be a null state. This means that the plane is disabled and will no longer participate in the generation of actions.

(3) The transition freeze is lifted, the pending transitions performed, and the modified agent continues its existence.

# 4 Surgery techniques

The framework presented previously permits almost arbitrary modifications in the structure of the agent. Without some self-discipline however, the modified agent will quickly become a chaotic assembly of active components. A successfull surgical operation is composed on a number of more disciplined elementary operations, with a well specified semantics. In these section we enumerate those techniques which we consider as being the most useful.

## 4.1 Simple surgical operations

The simplest surgical operations are referring to the adding and removal of individual states and transitions.

```
add state S [with stategy X];
remove state S;
add transition T;
remove transition T;
```

These operations are executed unconditionally. In the case of direct specification of operations is that the writer of the surgical blueprint must have a good knowledge of the existing agent structure. If new states are added to an existing plane, but not linked to existing states with transitions, they will never be executed. In order to remove existing states, transitions we need to know the name and function of the given states and transitions.

## 4.2 Replacing the strategy of a state

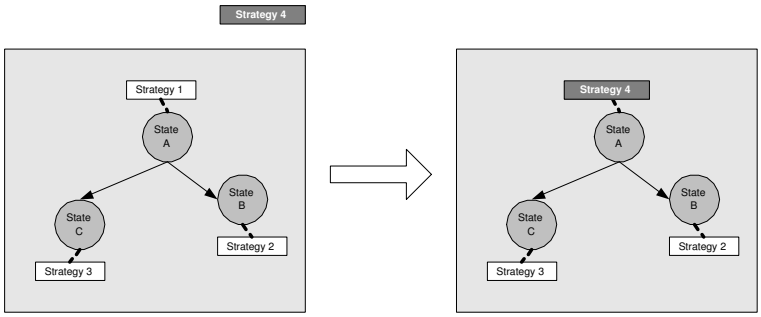


Figure 2: Replacing strategies

In this operation the strategy of a state is replaced with a different strategy. The reason of doing this is to improve or adapt the functionality to the specific condition of an agent. For example if an agent is running on, or migrated to a computer which doesn't have a graphical user interface, the strategies implementing the graphic user interface should be replaced with strategies adapted to the specific host, for example with command line programs. This operation is presented in Figure 2.

There is direct support in `blueprint` for these kind of operations:

```
change strategy OldStrategy on state State to NewStrategy;
```

The value of State can be *, meaning to replace all strategies of that type. These operations keep the old strategy namespace in which the strategy operates. In certain cases we should replace a group of interrelated strategies at once. For example, running and controlling an external application locally in Bond is done using the `Exec` strategy group. These strategies allow starting, supervising, terminating local applications, but can not run applications remotely. Now if the application

requires remote run, we can replace all these strategies with the corresponding strategies from the RExec strategy group, which run applications using the Unix rexec call. Then we do:

```
modify agent
  change strategy Exec.Start on state * with RExec.Start;
  change strategy Exec.Supervise on state * with RExec.Supervise;
  change strategy Exec.Terminate on state * with RExec.Terminate;
end modify.
```

Observe, that this surgical blueprint does not assume anything about the structure of agents.

## 4.3 Splitting a transition with a state

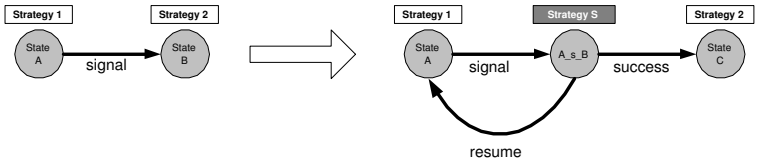


Figure 3: Splitting a transition with a new state

Transitions in the Bond system represent a change in the strategy. One important way of changing the functionality of an agent is by inserting a new state in a transition. There is direct `blueprint` support for this operation.

```
split transition TransitionLabel with strategy Strategy;
```

The effect of this transition is illustrated in Figure 3. In effect, the existing transition is redirected to the new state, while from the new state the "success" transition is generated to the original target. The name of the new state is generated automatically. This can be later used to add new transitions to the state.

Typical application of these techniques are:

-**Logging and monitoring**. In these case the inserted strategy writes the status to the log or sends a message to the monitor and immediately succeeds.

-**Confirmation checks.** The new strategy displays a confirmation dialog box (or verifies the confirmation using a different method). In the transition is confirmed, it succeeds, otherwise generates a `restore` transition, going be to the previous state. **Security checks** can be implemented in the similar way.

-**Synchronization.** Interagent synchronization can be implemented by an inserted strategy which waits for a synchronization message to perform the transition. This approach is important when.

## 4.4 Bypassing states

Bypassing a state is the semantic equivalent of replacing its strategy with a strategy which immediately, unconditionally succeeds. Practically the state is deleted, while all the incoming transitions are redirected to the target of the "success" transition originating from the given state.

The bypass operation can be used to revert the effect of the split operation discussed previously.

```
bypass state X; // or
bypass states having strategy Strategy;
```

For example, one application of the bypass operation is to remove the debugging or logging states from an agent.

## 4.5 Adding and removing planes

The planes of the multiplane finite state machine are expressing parallel activities. Thus, new functionality can be easily added to an agent by creating a new plane which implements that functionality. The new functionality can use the model of the world in the agent (the knowledge accumulated by other planes) and can directly communicate with other planes by triggering transitions in them. Analogously, we can remove functionality by deleting planes.

There is direct support in *blueprint* for adding and removing planes. A definition of a new plane in a surgical blueprint creates a new plane if there was no plane with the given name:

```
plane newplane
  // plane specification
end plane;
```

A plane can be removed with the following instruction.

```
delete plane name;
```

Of course, like any surgical operation adding and removing planes is a delicate operation. Generally, we can safely add and remove planes which represent an independent functionality. Generally, the planes added to an already working agent can be safely removed.

Examples of using this technique are:

-**Adding a visualization plane.** Visualization planes are presenting a part of the knowledge of the agent, i.e. the model in a visual format. They may be used to interactively modify the model. As these planes do not interfere with the functioning of the agent, they can be added and removed safely.

-**Adding a remote control/negotiation plane.** Agents which do not have remote control or negotiation properties, can be extended with a plane containing these features. External entities will communicate only with this plane, while the results will be communicated to the agent through the model or through transitions.

-**Replacing a reasoning plane.** Agents which perform logical reasoning are usually using a part of their model as a knowledge base of logical statements, and one of the planes is a reasoning plane, performing forward or backward reasoning on these statements. This reasoning plane can be replaced dynamically with a different one. For example, a performant external engine may have become available, thus the new reasoning plane can act only as a gateway, instead of providing the functionality itself.

## 4.6 Joining and splitting agents

Two of the simplest surgical operations on agents are the joining and splitting. When **joining** two agents, see Figure 4, the multi-plane state machine of the new agent contains all the planes of the two agents and the model of the resulting agent is created by merging the models of the two agents. The safest way is to separate the two models (for example through use of namespaces), but a more elaborate merging algorithm may be considered. As our design does not specify the knowledge representation method, the best approach should be determined from case to case. The agenda of the new agent is a logical function (usually an "and" or an "or") on the agendas of the individual agents. It is tempting to consider the joining of agents as a boolean operation on agents, and maybe to envision an algebra of agents. While the subject definitely justifies further investigation, the design presented in this paper do not qualify for such an algebra. The more difficult problem is handling the "not" operator, which applied to the agenda would render the current multi-plane state machine useless.

In case of agent **splitting** we obtain two agents, the union of their planes gives us the planes of the original agent (Figure 5). The splitting need not be disjoint, some planes (e.g an error handling or housekeeping) may be replicated in both agents. Both agents inherit the full model of the original agent, but the models may be reduced using the techniques presented in section 4.7. The agendas of the new agents are derived from the agenda of the original agent. The conjunction or disjunction of the two agendas gives the agenda of the original agent.

There are several cases when joining or splitting agents are useful: (a) Joining control agents from several sources, to provide a unified control, (b) Joining agents to reduce the memory footprint

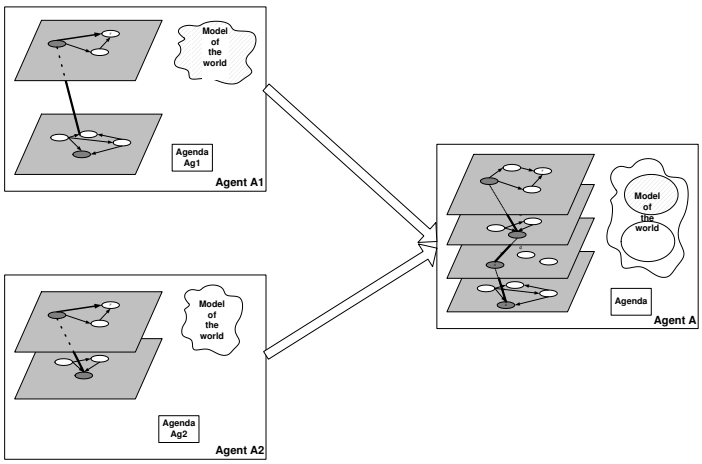


Figure 4: Joining agents: the new agent contains all the planes of the initial agents and a combination of their models. The agenda, in this case is the conjunction of the agendas of the original agents.

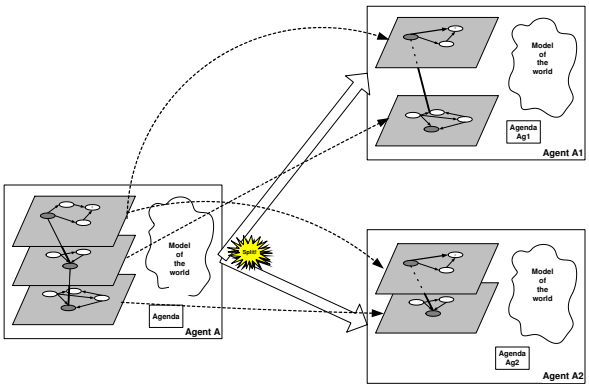


Figure 5: Splitting agents: the new agents contain a part of the planes of the original agent - in our case one plane is replicated in both new agents. The agents inherit the full model of the original agent while the conjunction of their agendas is the agenda of the original agent.

by eliminating replicated planes, (c) Joining agents to speed up communication, (d) Migrating only part of an agent, (e) Splitting to apply different priorities to parts of the agent.

Joining and splitting of agents is used by our implementation of agents implementing workflow computing.

## 4.7 Trimming agents

The state machines describing the planes of an agent may contain states and transitions unreachable from the current state. These states may represent execution branches not chosen for the current run, or states already traversed and not to be entered again. The semantics of the agent does not allow some states to be entered again, e.g. the initialization code of an agent is entered only once.

If the implementation allows us to identify the set of model variables which can be accessed by strategies associated with states, we can further identify parts of the model, which can not be accessed by the strategies reachable from the current state. The Bond system uses *namespaces* to perform a mapping of the model variables to strategy variables, thus we can identify the namespaces which are not accessed by the strategies reachable from the current state vector.

If the agenda of the agent can be expressed as a logical function on model variables and this is usually the case, we can simplify the agenda function for any given state, by eliminating the "or" branches that cannot be satisfied from the current state of the agent.

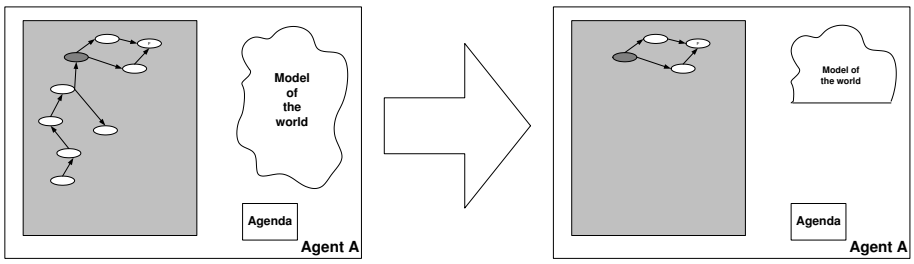


Figure 6: Trimming agents: the example presents the trimming of a single plane agent. The parts of the state machine which are not reachable from the current state are eliminated. Also the part of the model which can not be accessed by the remaining strategies is eliminated too.

All these considerations allow us to perform the "trimming" of agents, for any given state to replace the agent with a different, smaller agent as shown in Figure 6. In this example we used an agent with a single plane, but the process is identical for multi-plane agents. Both the multi-plane state machine, the model and the agenda can be simplified.

While stopping an agent to "trim" it is not justified for every situation there are several cases when we consider it to be useful:

- **Before migration.** As migration is sometimes proposed in order to "bring the code to the data" when the data is larger than the code, trimming the agent allows us to reduce the amount of code and internal data transferred. Furthermore, as for migration the agent has to be stopped anyhow, the penalty for the trimming time is usually compensated by the fact that less data and code needs to be transferred over the network.
- **Before checkpointing.** Similar to the migration case, the agent has to be stopped for checkpointing, and trimming can reduce the amount of data to be saved.
- **At runtime.** Trimming can be performed even on running agents, for example by an external thread or by a strategy of the agent. Doing this can be justified by the need to reduce the memory footprint of the agent. A careful examination is needed to find out whether the eliminated parts are actually becoming free memory. For compiled languages like C or C++ it is probably impossible to free the code memory, but even Java was not being able to garbage collect classes as of version 1.1. Freeing data implies different approaches whether the language has a garbage collector or not.

Our design implements just the framework and mechanisms for agent trimming. Determining the parts which can be trimmed is a problem in itself and various techniques can be used. Trimming the multi-plane state machine can be done by reachability analysis. Trimming the model depends very much on the specific implementation of the strategies. The Sethi-Ullman algorithm for reusing temporary variables from the theory of compiler construction [13] may be used. Trimming the agenda can use techniques from the theory of lazy evaluation of boolean expressions.

Generally this method can be considered as an extension of various techniques already in use in compiler and programming language theory to a different level of granularity, strategies instead of individual instructions. While the technique remains the same, the different granularity produces a shift in the cost/benefit analysis: there is not enough benefit in freeing an individual instruction from the memory, but it may be worthwhile for a strategy consisting of a large block of code.

The default migration implementation in the Bond system is using trimming to reduce the amount of data transferred in the migration process.

# 5 Realistic applications of agent surgery

While most researchers are skeptical about the applications of agent surgery, on a closer look these skepticism almost always have its roots in the technical feasability of the techniques. Once these problems are solved, the possibilities are very large.

Probably the most immediate applications of agent surgery will be

-the programs which are currently consisting of a exceedingly large number of features, out of which only a small number of is used by any particular user ("bloating").

-the cases where the same task is executed by different programs under different operating conditions. These programs have different user interfaces, security features etc, but they also contain significant replicated parts. This leads to independent configuration options, interoperability problems, which has to be solved every time one of the applications is upgraded.

To show how these problems can be handled by using surgically modified agent, let's consider a *personal information agent* used by an employee of a company. This agent encapsulates a number of functions: it accesses the email of the user, receives it's fax messages, pager calls, can process internet phone calls. It may include various filters, autoresponders, forwarders, loggers etc.

All these functionality is readily available in form of API-s, pluggable third party applications and libraries, and it can be assembled into an agent.

In the **office environment** the user uses a large screen desktop computer, with an operating system providing a graphical user interface. All components are accessible from here, and messages will be displayed on the screen, no matter how long they are or where they are coming from.

The network connection of the desktop computer is the company's intranet. While there is a password based authorization system, messages are usually not encrypted in the intranet. The security of the system is assured by firewalls, proxies and generally handled at company level. Thus, in this incarnation the agent does not need special security planes.

Let's assume now, that the user is accessing it's personal agent from its notebook computer, which is connected through a dialup ISP connection. The user interface is essentially the same, but the security situation is different. A special authentication procedure is needed, a different, secure access method should be used. One way of doing this is by using a secure tunnelling protocol like PPTP, but other methods can be employed. Certain applications might not be available from a dialup access.

If the professional is in a bussiness trip, he will download its personal agent in the rental car's computer. The network connection is done through wireless access and the visual interface is limited. Parts of the existing functionality of the agent can not be accessed from here. The visual user interface is replaced with a voice based interface, which can read the subject lines of incoming mails. To allow this, the mail filter is modified, while in the office environment basically only the unwanted junk-mail is filtered out, in this environment only the most urgent messages will be transmitted and read.

The personal agent can be accessed from devices like a street **communication booth**. The security problems with these kind of devices are even more difficult to solve that in the case of a dialup access, because in this case we don't have a trusted device. According to the company's internal policy, only some part of the personal agent may be downloaded to these devices and only some limited services accessed.

On a **handheld device** we have a visual interface with limited capability, where a different user interface has to be used. Certain features may not be available on a handheld either because of the limitations of the output capabilities, or because of the slowness of network connection. The network connection may be through wireless access (in which case it is probably slow) but it can be through a modem or even network card. In any case, what is almost sure, is that a handheld requires a different agent, than a desktop or notebook computer.

We can question if these situations really need surgery. Wouldn't just relying on Java based applications be enough? Java is platform independent, after all. Well, the platform independence of Java refers to the possibility of running the executable in various platforms, and as such, it provides the base of the previous scenario. But this feature, in itself do not solve the problems like:

-**Different user interface**: while Swing is portable, it can not run if there is no graphic interface - like in a voice based system, or if the interface is limited in resolution.

-**Different access methods**: the security issues.

-**Different intrinsic possibilities of the devices**: there will be always devices of different memory, computing power and so on. Programs have to adapt to these possibilities, not by going to the lowest common denominator but by choosing the best approach.

-**The unexpected:** an unexpected situation might prompt the company to completely change its access methods.

# 6 Conclusions

A number of factors, some of them technical, others legal and market-driven motivate an increasing interest in self-modifying programs. In this paper we argue that self-modifying or mutable programs are already around, and they will be an important component of the computing culture in coming years. In particular, this approach can open new possibilities for the field of autonomous agents.

Some of the applications of software agents require that the functionality be changed while the agent is running. To support efficiently agent mobility we propose to trim out all the unnecessary components of an agent before migrating it. But to change the functionality of an agent we have to modify its structure. This motivates our interest for agent surgery and by extension for program mutability.

Programs can be successfully modified if there is a high level, well-documented data structure that in some sense is analogous with the genetic information of biological structures. If the program designer chooses to have only part of the code described by this structure (like in the case of plugin API-s) weak mutability is possible. If the entire program is described by the data structure then strong mutability is possible. This is the case of Bond agents whose behavior is based on the multiplan state machine and agent surgery enables strong mutability.

We argue that several requirements must be met before mutabilility becomes an accepted programming technique. Mutability should have a well-defined scope, be self-sufficient, seamless, persistent, and reversible. For example we should be able to modify running applications because it is unconceivable to stop a critical system to modify its components.

The Bond agent framework is distributed under an open source license (LGPL) and the second beta release of version 2.0 can be downloaded from `http://bond.cs.purdue.edu`.

# 7 Acknowledgments

The work reported in this paper was partially supported by a grant from the National Science Foundation, MCB-9527131, by the Scalable I/O Initiative, and by a grant from the Intel Corporation.

# References

[1] A. Clements *Glossary of computer architecture terms. Self-modifying code.* `http://www-scm.tees.ac.uk/users/a.clements/Gloss1.htm`

[2] Here comes a reference to code mangling

[3] REference to self building neural networks

[4] reference to dynamic workflows

[5] L. Bölöni, R. Hao, K.K. Jun, and D.C. Marinescu, *Structural Biolgy Metaphors Applied to the Design of a Distributed Object System*, Proc. Second Workshop on Bio-Inspired Solutions to Parallel Processing Problems, In LNCS, vol 1586, Springer Verlag, pp. 275-283, 1999.

[6] L. Bölöni and D.C. Marinescu "Biological Metaphors in the Design of Complex Software Systtems", *Journal of Future Computer Systems*, Elsevier, 1999, (in press)

[7] P. Tsompanopoulou, L. Bölöni and D.C. Marinescu *The Design of Software Agents for a Network of PDE Solvers* Agents For Problem Solving Applications Workshop, Agents '99, IEEE Press, pp. 57-68, 1999.

[8] L. Bölöni and D. C. Marinescu. *A Multi-Plane State Machine Agent Model* Technical Report CSD TR 99-027, Purdue University, September 1999.

[9] K. Palacz and D. C. Marinescu, *An Agent-Based Workflow Management System* Computer Sciences Department, Purdue University, CSD-TR#99-032, October 1999.

[10] K.K. Jun, L. Bölöni, D. K.Y. Yau, and D. C. Marinescu, *Intelligent QoS Support for an Adaptive Video Service* Computer Sciences Department, Purdue University, CSD-TR#99-033, October 1999.

[11] K.K. Jun, L. Bölöni, K. Palacz and D. C. Marinescu, *Agent-Based Resource Discovery* Computer Sciences Department, Purdue University, CSD-TR#99-034, October 1999.

[12] L. Bölöni and D.C. Marinescu *A Multi-Plane State Machine Agent Model* Purdue University CSD-TR #99-027

[13] R. Sethi and J. D. Ullman. *The generation of optimal code for arithmetic expressions* Journal of the ACM, 17(4):715-728, October 1970.

[14] *A gentle introduction to Bond* `http://bond.cs.purdue.edu/guide/Intro.ps`

[15] T. Finin, et al. *Specification of the KQML Agent-Communication Language*, DARPA Knowledge Sharing Initiative draft, June 1993